Check for manifest misconfigurations

To read the manifest file and other files and assets you can use:

```
sudo aptitude install apktool
apktool d blabla.apk
cd res/xml
```

Allow backup set to true

In the manifest file it is possible to specify if it should be possible to backup the application data, if the user has usb debugging enabled. Applications that store sensitive data on disk should have this set to false:

Check for the following in the manifest file:

```
android:allowBackup="false"
```

If backup is allowed you can backup the data like this:

```
adb backup -apk com.theapp.blabla
```

```
sudo aptitude install qpdf #this is where zlib-flate is found
dd if=backup.ab bs=24 skip=1 | zlib-flate -uncompress > res.tar
tar xvf res.tar
```

Check if application is run in debuggable mode

If the application is configured to run in debuggable mode the following will be set in the manifest file.

```
<application debuggable="true"
```

It can also be checked with drozer with:

```
run app.package.debuggable
```

Check network security configurations

Cleartexttraffic is set to false by default in android 9.

Network traffic configuration is set in the manifest-file.

If you see this in the manifest-file it means that the configurations are made in the network-security file.

```
android:networkSecurityConfig="@xml/network_security_config"
```

The network-security file is found in `res/xml/network_security_config.xml`.

Check if `domain-config cleartextTrafficPermitted="true"` is set to true, and to what domain it is allowed.

Check permissions

If the application has to wide permissions it might be a concern for privacy, but usually not a vulnerability or risk to the application itself.

Check permissions i the manifest file or with Drozer.

Drozer command:

```
run app.package.info -a com.yourapp.blabla
```

There is no general right or wrong. Are the permissions correct in relation to what the application is doing?

Some a universally dangerous permissions, like the following.

There is not many legitimate reasons to allow an app to install other packages.

```
android.permissions.INSTALL_PACKAGES
```

Each permission has a android-defined permissions level: `normal`, `dangerous` and `signature`.

https://developer.android.com/guide/topics/permissions/overview#normal-dangerous

If a permission is defined as "dangerous", such as reading contacts. The user must explicitly agree to the permission for it to be active.

Exploiting application components

There are four types of "components" in an android applications: - Activity - screens/views - Content provider - database objects - Services - background workers - Broadcast receiver

Three of these four components are activated using asynchronous messages called "intents". The components that are activated using intents are `activity`, `services`, and `broadcast receivers`

If a component is exported or not is defined in the `manifest.xml` file.

You can map the attacksurface of the components using these drozer commands

```
run app.package.list

run app.package.attacksurface app.theapp
```

Exploiting exported activities

What is an intent?

"An Intent provides a facility for performing late runtime binding between the code in different applications. Its most significant use is in the launching of activities, where it can be thought of as the glue between activities. It is basically a passive data structure holding an abstract description of an action to be performed."

A good explanation of how this attack works is here: https://appsec-labs.com/portal/hacking-android-apps-through-exposed-components/

Activities that are specified as "exported" in the manifest file can be run by other applications on the phone. In a sense, the other application is just running a function on the victim-application. What that function does depends on the application. The Activities might just start a new view. But they may also include arguments.

Look for activities that are exported and allows arguments.

```
.getExtras
```

To check which activitites are exported run:

```
run app.package.list
run app.package.attacksurface com.appen.blabla

Attack Surface:
  2 activities exported
  2 broadcast receivers exported
  0 content providers exported
  2 services exported
```

Now you want to check which activities those are:

```
run app.activity.info -a com.appen.blabla
```

```
run app.activity.start --component com.mwr.example.sieve com.mwr.example.sieve.PWList
```

Hidden view

There might be hidden functionality in the application, than can be accessed by a user, using adb or drozer. It might be the case that only premium users should be able to access certain functionality. ANd that functionality is accessible in the application, but there is no backend checks. So a user would be able to use the free version, but access functionality that should only be accessible to premium users.

Check for:

List the activities, and test them. Do they expose functionality that should not be accessible to the user.

```
run app.activity.info -a com.theapp.blabla
```

Phishing

Check if an activity that is exported, and therefore can be started by other applications, can be used in some kind of phishing-attack. The idea would be to create a fake application, that does something, and then open up the activity of the victim-application, thereby tricking the victim into inserting data into that view.

Check for by starting the different views and see if any can be used in phishing attacks.

```
run app.activity.info -a com.theapp.blabla
```

Injections

Activities can be configured to accept arguments. This means that a malicious application could start the activity and also insert arguments that might be processed in some way. An activity is just an event, and can be programmed to do whatever, so you will need to review the source-code of the activity in order to see if it takes arguments, and in that case see how the argument is processed. Maybe the argument is used to access an sql-database on the phone, or something else like that.

Activities that have arguments use the `getExtras()` function. So search the source-code for `getExtras`, and see if it is used with any of the exported activities. You can also look for `getStringExtra()`.

If you find one you can use it like this:

```
dz> run app.activity.start --component  app.theapp.dev app.theapp.core.main.theactivity --extra string aaaaa bbb
```

Exploiting insecure Content Providers

Content providers are interfaces for sharing data between applications. Each content provider has en url that looks something like this `content://`. Other applications can use the content provider to query information. They can use functions like `insert()`, `query()`, and `update()` to access data in another application. If this is configured incorrectly another application could possibly access sensitive data from the application.

The data might be a file written to the disk, or it might be written to a database, or some other persistent storage.

If content providers are exported

Again content providers that are exported means that they are exposed to other applications. So, in order to map the exported content providers you can look in the manifest file or check it using drozer. With drozer it looks like this:

```
run app.package.attacksurface app.theapp
```

If you find that there are some content providers that are exported you can get more info on them with this command:

```
run app.provider.info -a app.theapp
```

To list all providers, even the content providers that other application can't access you add the `-u` flag.

```
run app.provider.info -a app.theapp -u
```

Check if sensitive information can be accessed

Check for uri paths:

```
run app.provider.finduri com.appen.blabla
```

If you find some content providers that are exported you can query those and see if you can access some sensitive information. This can be done like this:

```
run app.provider.query content://com.theapp.uripath/
```

Check for SQL injections

Content providers are sometimes connected to a SQLite database.

Check out https://solidgeargroup.com/en/sql-injection-in-content-providers-of-android-and-how-to-be-protected/ resources how to check for this and exploit.

Attacking Insecure Services

A services is a component that is running in the background to perform long-running operations or to perform work for remote processes.

A service might play music in the background while the user is in a different application. Or it might access data over the network while the user is doing something else. There are three different types of services `scheduled`, `started`, and `bound`.

The services are listed in the manifest file, it looks something like this in the manifest:

```
<service android: name=".Example Service" />
```

```
run app.service.info -a com.theapp.blabla
```

You can run a service like this:

```
run app.service.start --action com.theapp.blabla.somService --component com.theapp.blabla com.theapp.blabla.somService
```

https://book.hacktricks.xyz/mobile-apps-pentesting/android-app-pentesting/drozer-tutorial

Attacking broadcast receivers

Applications can register broadcast receivers, this will allow them to be able to receive broadcast from the OS or from other applications. It is like a publish-subscribe pattern. If an application registers to receive a specific broadcast the system till route traffic to that application when such a broadcast is published.

Storage

Data can be stored in multiple ways in an android device. For example:

- Shared Preferences
- SQLite Databases
- Realm Databases
- Internal Storage
- External Storage

Check for sensitive information in shared_prefs

The SharedPreferences API is commonly used to permanently save small collections of key-value pairs. Data stored in a SharedPreferences object is written to a plain-text XML file. The SharedPreferences object can be declared world-readable (accessible to all apps) or private. Misuse of the SharedPreferences API can often lead to exposure of sensitive data.

Check the following file to see if sensitive data (like passwords or tokens) are stored in it.

```
/data/data/<package-name>/shared_prefs/key.xml
```

Check for world readable files in the app

Maybe som files on the application for some reason has received incorrect filepermissions, drozer can find those:

```
run scanner.misc.readablefiles /data/data/com.theapp.blabla
```

Or you can go to the app:

```
ls -la /data/data/app.theapp.blabla/shared_prefs

And look for
-rw-rw-r--
```

From the owasp guide: "Please note that `MODE_WORLD_READABLE` and `MODE_WORLD_WRITEABLE` were deprecated starting on API level 17. Although newer devices may not be affected by this, applications compiled with an android:targetSdkVersion value less than 17 may be affected if they run on an OS version that was released before Android 4.2 (API level 17)."

Check for unencrypted sensitive files in SQL database

Databases are stored in `/data/data/[packeage name]/databases/` .

You can usually not pull them down using adb directory with `pull` because you need to be root. So instead you can move the data to sdcard.

```
adb shell
mkdir /sdcard/exfil
cp /data/data/[packeage name]/databases/* /sdcard/exfil

adb pull /sdcard/exfil
```

```
sudo aptitude install sqlite

sqlite3 database.db
.dump
```

Check for sensitive information in external storage

External storage are storage that may be removable, such as an sdcard, or built in. These storages are world reaible. So if the application stores data here any other

application can read that data.

Also, files stored outside of the `/data/data/[package name]/` are not removed if the application is uninstalled.

How are creds stored?

Connect your device to to your pentest-vm. Install ADB.

Run

```
adb shell
To become root:
su
```

Now you can go to /data/data/com.yourappliucation.test

Check

Misc

Check for hardcoded credentials

Check for strings. MobSF lists all strings.

Unintended data leaks

Check that sensitive data is not logged

Use adb and get a shell. Run `logcat` as root. And then use the applicaiton, do you see passwords, tokens, or other sensitive data?

You can also check the source code for logs.

Disallow copy-paste for sensitive data

The clipboard can be accessed by all applications on the phone. It should not be possible to copy data that is considered sensitive, as that might leak to other applications.

Analytics sent to third party

Check that sensitive data is not leaked to third parties

Check for Firebase access control

Look for a string looking like this `somethingsomething.firebaseio.com`. Check if you can access sensitive data in: `somethingsomething.firebaseio.com/.json`

Webview attacks

Check that javascript is disabled

Javascript is disabled by default. But can be enabled with:

```
setJavaScriptEnabled()
```

Network security

Check for non-http traffic

When the device is on your access-point, run wireshark and dump its traffic. Do you see anything unusual that might indicate that the application is communicating over non-http?

Check for unencrypted http traffic

Use the application. Check in burk or wireshark that no non-tls-traffic is sent.

Check if application verifies server certificate

Two key issues need to be verified. - That the application verifies that the server certificate comes from a trusted CA - Determine weather the endpoint server

presents a the right certificate

```
openssl s_client -proxy localhost:8080 -connect google.com:443 -showcerts
```

Verify that self-signed certificates are not accepted

In burp go to `Proxy / Options / Proxy Listener / Edit / Certificate / Use Self signed certificate` Then try to use the application, if the application is still working and you see requests going through then the application is not checking the servers certificate.

Verify that correct hostname certificates are not accepted

It might be that the developers made the application check for a specific hostname in the CN-field in the server certificate, but nothing else

This can be done automatically using burp if you go to `Proxy / Options / Proxy Listener / Edit / Certificate / Generate CA Signed certificate with specific hostname` HOWEVER, this will make burp generate a certificate from the CA portswigger, so if you have already installed your portswigger root certificate in your mobile phone the phone will trust your certificate. Which of course is by design and not an issue. So only use the burp method if you have removed the burp CA in your phone.

Otherwise you can just generate your own certificate, with this oneline. Remember to put in the correct CN (Common name), it should be the common-name that is the same as the one that the legitimate server presents.

```
openssl req -newkey rsa:2048 -nodes -keyout key.pem -x509 -days 365 -out certificate.pem
```

Check how your certificate came out with this command:

```
openssl x509 -text -noout -in certificate.pem
```

Now you have to combine your private key with your public certificate. The archive/format for this is pkcs12. And the combined archive can be create using the following command:

```
openssl pkcs12 -inkey key.pem -in certificate.pem -export -out certificate.p12
```

Set a password

Now go to Burp `Proxy / Options / Proxy Listener / Edit / Certificate / Use a Custom Certificate` Add your pkcs12 formatted certificate and key, insert the password. Rememer, the private key is used to decrypt the traffic. Without

Check that you have actaully done everything right and download the certificate

```
openssl s_client -proxy localhost:8080 -connect PAGEYOURARETESTING.com:443 -showcerts
```

Now use the application and see if it works. If you can see the traffic it means the mobile application accepts your crappy self-generated certificate, and ca therefore be performed by an attacker.

If the application does not work, it does not send any requests through burp, everything is working as it should.

References

https://labs.f-secure.com/assets/BlogFiles/mwri-drozer-user-guide-2015-03-23.pdf